

University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, &
Professional Papers

Graduate School

1993

Reliable file locking manager and monitor in a client/server distributed system environment

Kuang-hsin Lin

The University of Montana

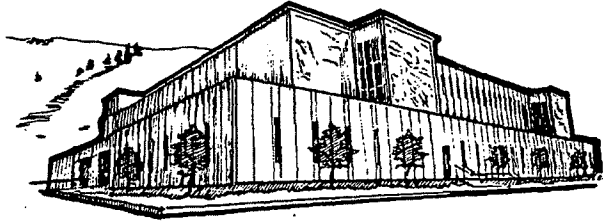
Follow this and additional works at: <https://scholarworks.umt.edu/etd>

Let us know how access to this document benefits you.

Recommended Citation

Lin, Kuang-hsin, "Reliable file locking manager and monitor in a client/server distributed system environment" (1993). *Graduate Student Theses, Dissertations, & Professional Papers*. 5095.
<https://scholarworks.umt.edu/etd/5095>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike MANSFIELD LIBRARY

The University of
Montana

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

**** Please check "Yes" or "No" and provide signature****

Yes, I grant permission X
No, I do not grant permission —

Author's Signature Li Kuang Hsin

Date: 4-12-93

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

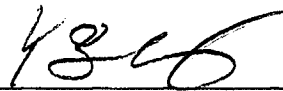
A Reliable File Locking Manager and Monitor in
a Client/Server Distributed System Environment

Kuang-Hsin Lin

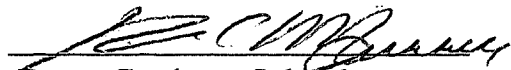
Department of Computer Science
University of Montana

Presented in partial fulfillment of the requirements
for the degree of
Master of Computer Science
University of Montana
October 12, 1993

Approved by



Chairman, Board of Examiners



Dean, Graduate School

Nov. 16, 1993

Date

UMI Number: EP40559

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP40559

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Lin, Kuang-Hsin., M.S., October 1993

Computer Science

A Reliable File Locking Manager and Monitor in a Client/Server
Distributed System Environment (49 pp.)

Director: Dr. Youlu Zheng



ABSTRACT

The focus of this project is to develop two server programs which are Network Lock Manager (NLM) and Network Status Monitor (NSM) as well as a client program to provide network file and record locking services in a client/server distributed system environment.

NLM and NSM are Remote Procedure Call (RPC)-based services that normally execute autonomous "*daemon*" services on Network File System (NFS). They work together to provide a reliable file and record locking over NFS.

The client program which works as a communicating interface between application programs and the server programs NLM and NSM is also an RPC-base service. The functions *rflock()* and *rfcntl()* in the client program provide the exactly same services as the functions of UNIX system calls *flock()* and *fcntl()*, and extend the file locking capability over NFS in a reliable manner.

TABLE OF CONTENTS

Abstract	ii
Table of Contents	iii
List of Figures	v
Chapter 1 Introduction and Overview	1
1.1 Client Server Model	1
1.2 Background	1
1.3 Remote Procedure Call (RPC) Overview	2
1.4 Project Approach	4
1.5 NLM Overview	5
1.6 NSM Overview	6
1.7 System Failure	8
Chapter 2 NLM Protocol	10
2.1 Introduction	10
2.2 NLM RPC-based Procedures	10
2.3 Semantics of NLM Operations	11
Chapter 3 NSM Protocol	18
3.1 Introduction	18
3.2 NSM RPC-based Procedures	18
3.3 Semantics of NSM Operations	19
Chapter 4 System Analysis	22
4.1 Concurrent Or An Iterative Design	22
4.2 Design A Reliable File Locking Service	23
4.3 A Better Design	25
Chapter 5 System Design and Implementation	26
5.1 Implementation Approach	26
5.2 RPCgen	26
5.3 Designing the Server	27
5.3.1 Six Steps To Build A Server	27
5.3.2 Implement NLM As A Concurrent Server	30
5.3.3 Implement NSM As An Iterative Server	31
5.4 Designing The Client	31
5.5 The Interaction Between The NLM and The NSM	34
5.6 How An RPC System Work	36
5.6.1 RPC Library Calls	37
5.6.2 Using RPC Library Calls in a Program	38

Chapter 6 Conclusion	40
Bibliography	41
Acknowledgment	42
Appendix 1	43
Appendix 2	45
Appendix 3	47

LIST OF FIGURES

Figure 1-1 4
Figure 1-2 7
Figure 4-1 22
Figure 4-2 23
Figure 5-1 28
Figure 5-2 33
Figure 5-3 36
Figure 5-4 39

CHAPTER ONE

Introduction and Overview

1.1 Client Server Model

The *client/server* paradigm is a method to allow a programmer to establish communication between two application programs and to pass data back and forth, and these two application programs can be executed on the same machine or on different machines.

The *client/server* paradigm divides communicating applications into two broad categories, depending on whether the application waits for communication or initiates it. In general, an application that initiates communication is called *client*. End users usually invoke client software when they use a network service. Most client software consists of conventional application programs. Each time a client application program executes, it contacts a *server*, sends a request, and awaits a response. When the response arrives, the *client* continues processing.

By comparison, a *server* is the program that waits for incoming communication requests from a *client*. The *server* receives a client's request, performs the necessary computation, and returns the result to the *client*.

1.2 Background

Network File System (NFS), developed by Sun Microsystems Incorporated (SUN) provides on-line remote access to shared file systems over various networks.

Theoretically, NFS is a kind of client/server communication but it hides all the complicated client/server operations from users so that from the end users' point of view, NFS is almost invisible. Once a remote file system has been mounted on a local file system, users access remote files using exactly the same operations as they use for local files. However, NFS is stateless, so that a server need not maintain any state information about the clients that it services. This means that a client is independently responsible for completing work, and that a server need not remember any thing from one call to the next. With no state left on the server, there is no state to recover when the server crashes and comes back up. So, from the client's point of view, a crashed server appears no different from a very slow server.

Because of this statelessness, NFS cannot provide inherently statefull services like file locking. Instead, this service is provided by two cooperating processes: the Network Lock Manager (NLM) and the Network Status Monitor (NSM). The NLM and NSM are Remote Procedure Call (RPC)-based services that normally execute as autonomous "*daemon*" services on both NFS client and server systems. They work together to provide stateful file and record locking over NFS. This philosophy was first introduced by SUN, and it has been adopted by most UNIX systems to provide network file and record locking services over NFS.

1.3 Remote Procedure Call (RPC) Overview

RPC allows a client to execute procedures on other networked computers or servers. RPC is also a user programming tool. It makes the client/server model more powerful and easier to program in than yesterday's low-level network socket interface.

Normally the RPC model can be divided into three categories

RPC specification

RPC protocol compiler - RPCgen

RPC library routines

The RPC specification provides a procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. NFS is one such "program". The combination of host address, program number and procedure number specifies one remote service procedure. RPC does not depend on services provided by specific protocols, so it can be used with any underlying transport protocol (TCP/IP or UDP/IP).

The RPCgen, an RPC protocol compiler, is the most important tool for writing distributed applications. It is used to generate the code, a client/server interface stub, to do the job that is described in the RPC specification.

The RPC library routines isolate the C programmer from the specialized data structure developed on top of UNIX IPCs (Inter Processes Communication) for the purpose of remote procedure calling. The basic process consists of the client finding the appropriate service, checking authentication, and placing a request to run a service procedure. After a client request is made and validated, the server replies to the client with results generated by the selected procedure.

OSI Layers

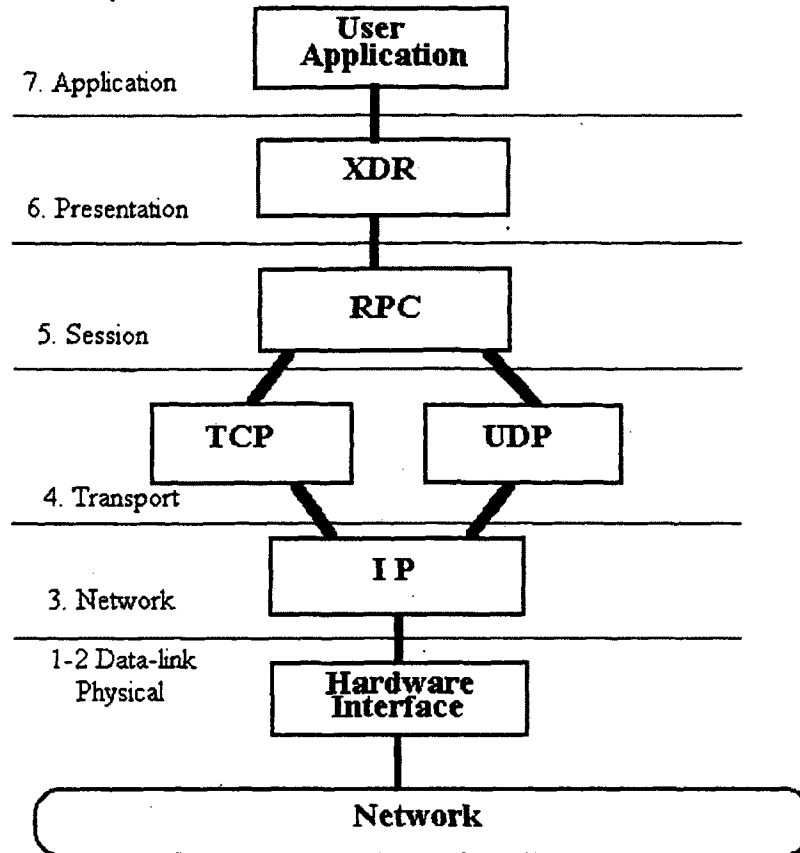


Figure 1-1. *RPC systems within the OSI reference model.*

1.4 Project Approach

The objective of this project is to design and implement NLM and NSM that are a RPC-based client/server model to provide a reliable network file and record locking services over the NFS.

In this project, the design and implementation of NLM and NSM are based on the well known protocol, SUN's NLM and NSM protocols, so that the NLM and NSM can accept all the clients' requests in different UNIX systems.

In addition to the NLM and NSM that work as a server, a client program that works as a communication interface between an application program and a server is included in this project.

1.5 NLM Overview

The NLM, called the lock *daemon* in a UNIX system, is a process running in the background providing advisory file and record locking in an NFS environment.

When a lock service is requested by a client, the server NLM will request a lock, which may be a file locking or a record locking on the local file system for the client. Once the lock is acquired, the server NLM will send a reply to the client indicating that the lock has been acquired. In addition to providing the locking service, the NLM also provides several functions such as lock-test which tests if a region is locked by another process, unlock which releases a lock, lock-cancel which cancels a blocked locking request and lock-reply which sends a reply back to the client. All these functions can be requested synchronously or asynchronously.

The client portion of an NLM may choose to implement any one of the functions provided by a server using either set of procedure calls.

For the synchronous requests, requests and replies are just like normal procedure calls. The caller (client) requests a service from the callee (server) and then waits for a reply from the callee.

For the asynchronous requests, the caller (client) does not wait for a reply after sending a request. After the server completes the requested job, the server will act as a client, requesting a "reply" procedure call from the client, which acts as a server.

1.6 NSM Overview

The NSM, called state *daemon* in a UNIX system, provides information on the status of network hosts. Each NSM keeps track of its own "state" and notifies any participating computers of a change in this state. We shall consider an example for clarity. Let us assume that a process on computer A has locked a region in a file system mounted over NFS on computer B. If B now receives a request for the same region from computer C, there are two possibilities: either B waits (under certain circumstances forever) until the end of A's process, or it checks whether A is crashed and hanging in the network. If this is the case, the locking process no longer exists and the region is immediately released to C (via B). Therefore, once one of the two connected machines crashes, all the regions locked by the crashed machine should be released. On the other hand, when the crashed machine reboots (re-starts), it should re-send locking requests to recover all the locks that were lost. The so-called NSM, network status monitor, exists to handle lock recovery and release when a connected machine has crashed.

The following figure (Figure 1-2) shows the execution of a lock operation in the network.

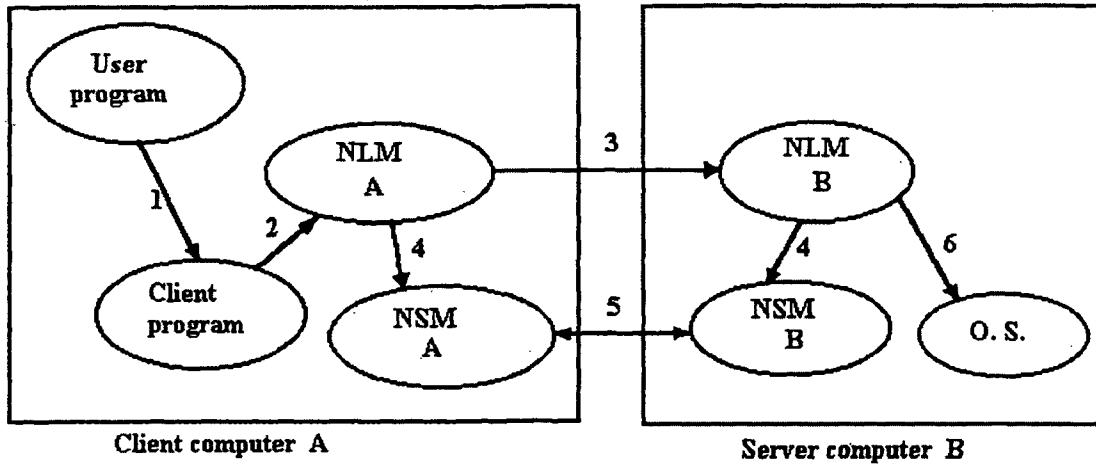


Figure 1-2. A lock operation in the network.

Let us examine the sequence of NLM and NSM operations in Fig. 1-2 in more detail:

- (1) A user program on computer A generates a request to the client program to lock a file.
- (2) The NLM on computer A receives a request from the client program to lock a file in computer B.
- (3) The NLM on client computer A uses an RPC request to ask the server computer B to lock a region of a file for it. In normal situations, a LOCK_OK signal will be returned to the client NLM on computer A.

- (4) (On Client computer A) After the client NLM receives a LOCK_OK signal from server NLM, it will use an RPC to request the local NSM to monitor the server computer B.
- (4) (On Server computer B) After the server NLM sends a LOCK_OK signal to the client NLM, it will use an RPC to request the local NSM to monitor the client computer A.
- (5) Both NSMs are monitoring each other.
- (6) The NLM on server computer B locks the file for the client computer A.

1.7 System Failure

As in other NFS services such as *read()* and *write()*, so too for network-wide locking: The first commandment is transparency to the local file system. After a server crashes and restarts, the lock service attempts to reproduce the status as it was before the crash, so that the application running on the client should notice nothing of the interruption other than a time delay. Of course, this is only possible for the server; failure of a client computer would also destroy the application program. In this case the server can and must release the lock of a process that has ceased to exist. In both cases, the partner involved knows nothing of the interruption until the computer that crashed is restarted.

After system (server) start-up, but before the NLM passes over to normal operation, the system waits for a short time called the grace period. During this time

interval, client computers have a chance to reinstall the lock that was lost when the system crashed. A new lock may only be requested after this waiting period.

CHAPTER TWO

NLM Protocol

2.1 Introduction

This chapter discusses the NLM and describes how NLM is defined to be a remote program using SUN RPC. The NLM contains 19 procedures providing advisory file and record locking. By invoking these procedures, the client can do all the locking related operations remotely from the server.

2.2 NLM RPC-based Procedures

The following specification summarizes the protocol used by the NLM using RPC language. The detailed data structures for NLM procedures are in appendix 1.

```
/* NLM procedures */

program NLM_PROG {
    version NLM_VERSX {

        /* synchronous procedures */
        void      NLM_NULL(void) = 0;
        nlm_testres NLM_TEST(struct nlm_testargs) = 1;
        nlm_res     NLM_LOCK(struct nlm_lockargs) = 2;
        nlm_res     NLM_CANCEL(struct nlm_cancargs) = 3;
        nlm_res     NLM_UNLOCK(struct nlm_unlockargs) = 4;
        nlm_res     NLM_GRANTED(struct nlm_testargs) = 5;

        /* sever NLM call-back procedure to grant lock */
        void      NLM_TEST_MSG(struct nlm_testargs) = 6;

        /* asynchronous requests and responses */
        void      NLM_LOCK_MSG(struct nlm_lockargs) = 7;
```

```

void    NLM_CANCEL_MSG(struct nlm_cancargs) = 8;
void    NLM_UNLOCK_MSG(struct nlm_unlockargs) = 9;
void    NLM_GRANTED_MSG(struct nlm_testargs) = 10;
void    NLM_TEST_RES(struct nlm_testres) = 11;
void    NLM_LOCK_RES(nlm_res) = 12;
void    NLM_CANCEL_RES(nlm_res) = 13;
void    NLM_UNLOCK_RES(nlm_res) = 14;
void    NLM_GRANTED_RES(nlm_res) = 15;

/* private procedures */
int     NLM_PRV_REG(struct lock_node) = 60;
int     NLM_PRV_RM(struct status) = 61;
int     NLM_PRV_RESEND(struct status) = 62;

} = 1;
) = 100021;

```

The NLM provides synchronous and asynchronous procedures that provide the same functionality. The server portion of an NLM implementation must support both the synchronous and asynchronous procedures.

The asynchronous procedures implement a message passing scheme to facilitate synchronous handling of locking and unlocking. Each of the functions Test, Lock, Unlock and Grant is separated into a message part and a result part. An NLM (parent process) will send a message to another NLM (child process) to perform some action. The receiving NLM will queue the request. When the request is de-queued and completed, the NLM will send the appropriate result via the result procedure. For example an NLM may send an NLM_LOCK_MSG and will expect an NLM_LOCK_RES in return. These functions have the same functionality and parameters as the synchronous procedures.

2.3 Semantics of NLM Operations

The following sections describe how each of the NLM remote procedures operates.

2.3.1 NLM_NULL (Procedure 0)

By convention, procedure 0 in any RPC program is termed *null* because it does not perform any action. An application can call it to test whether a given server is responding.

2.3.2 NLM_TEST (Procedure 1)

This procedure tests to see whether the lock specified by arguments is available to this client.

2.3.3 NLM_LOCK (Procedure 2)

This procedure supports two types of file region locking, which are BLOCK and NON-BLOCK. If the "BLOCK" is requested and the lock request cannot be granted immediately, the server will return a status of "LOCK_BLOCKED" for this procedure call. When the request can be granted, the server will make a call-back to the client with the NLM_GRANTED (procedure 5) procedure call. If the "NON-BLOCK" is requested and the lock cannot be granted immediately, no NLM_GRANTED call-back will be made.

2.3.4 NLM_CANCEL (Procedure 3)

This procedure cancels an outstanding blocked lock request. If the client made an NLM_LOCK procedure with "BLOCK" request, and the procedure was blocked by the

server, the client can choose to cancel this outstanding lock request by using this procedure.

2.3.5 NLM_UNLOCK (Procedure 4)

This procedure will remove the lock specified by the arguments.

2.3.6 NLM_GRANTED (Procedure 5)

This procedure is a call-back procedure from the server NLM (running on the host where the file resides) to the client. With this procedure, the server is the caller and the client is the recipient.

A client issuing an NLM_LOCK procedure that blocks will be returned a status of "LOCK_BLOCKED", indicating the lock cannot be granted immediately. At a later point, when the lock is granted, the server will issue an NLM_GRANTED procedure call to the client to indicate the lock has been granted.

2.3.7 NLM_TEST_MSG (Procedure 6)

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_TEST procedure.

2.3.8 NLM_LOCK_MSG (Procedure 7)

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_LOCK procedure.

Like NLM_LOCK, this procedure supports "BLOCK" and "NON-BLOCK" requests. If "BLOCK" is requested and the lock request cannot be granted immediately, the server will return an NLM_LOCK_RES (procedure 12) with a status of "LOCK_BLOCKED". When the request can be granted, the server will make a call-back to the client with an NLM_GRANTED_MSG (procedure 10) procedure. If "NON-BLOCK" is requested and the lock cannot be granted immediately, the server will return an NLM_LOCK_RES procedure with a status of "LOCK_DENIED" and no NLM_GRANTED_MSG call-back will be made.

2.3.9 NLM_CANCEL_MSG (Procedure 8)

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_CANCEL procedure.

If the client makes an NLM_LOCK_MSG procedure with a "BLOCK" request, and the procedure is blocked by the server, the client can choose to cancel this outstanding lock request by calling this procedure.

2.3.10 NLM_UNLOCK_MSG (Procedure 9)

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_UNLOCK procedure.

2.3.11 NLM_GRANTED_MSG (Procedure 10)

This procedure is one of the asynchronous RPCs. It performs the same function as the NLM_GRANTED procedure. Like NLM_GRANTED, it is a call-back procedure from the server NLM (running on the host where the file resides) to the client.

A client issuing an NLM_LOCK_MSG procedure that blocks will be returned a status of "LOCK_BLOCKED", indicating the lock cannot be granted immediately. At a later point, when the lock is granted, the server will issue an NLM_GRANTED_MSG procedure call to the client to indicate the lock has been granted.

2.3.12 NLM_TEST_RES (Procedure 11)

This procedure is one of the asynchronous RPCs. The server calls this procedure to return results of the NLM_TEST_MSG procedure to the client (the host issuing the NLM_TEST_MSG call).

2.3.13 NLM_LOCK_RES (Procedure 12)

This procedure is one of the asynchronous RPCs. The server calls this procedure to return the results of the NLM_LOCK_MSG procedure to the client (the host issuing the NLM_LOCK_MSG call).

2.3.14 NLM_CANCEL_RES (Procedure 13)

This procedure is one of the asynchronous RPCs. The server calls this procedure to return the results of the NLM_CANCEL_MSG procedure to the client (the host issuing the NLM_CANCEL_MSG call).

2.3.15 NLM_UNLOCK_RES (Procedure 14)

This procedure is one of the asynchronous RPCs. The server calls this procedure to return the results of the NLM_UNLOCK_MSG procedure to the client (the host issuing the NLM_UNLOCK_MSG call).

2.3.16 NLM_GRANTED_RES (Procedure 15)

This procedure is one of the asynchronous RPCs. The server calls this procedure to return the results of the NLM_GRANTED_MSG procedure to the client (the host issuing the NLM_GRANTED_MSG call).

2.3.17 NLM_PRV_REG (Procedure 60)

This procedure is one of the private synchronous RPCs. The procedure will be called by a local client program. Every time a client want to lock a remote file, the client should invoke this procedure to record the lock information in the local NLM server so that if the remote machine crashes the local server can reclaim the lock automatically.

2.3.18 NLM_PRV_RM (Procedure 61)

This procedure is one of the private synchronous RPCs. It is a call-back procedure to remove a lock and a lock node from a linked list.

2.3.19 NLM_PRV_RESEND (Procedure 62)

This procedure is one of the private synchronous RPCs. It is a call-back procedure that will reclaim all the locks in the given remote host.

CHAPTER THREE

NSM Protocol

3.1 Introduction

This chapter describes the NSM protocol that is related to, but separate from, the NLM protocol. The NSM protocol is not specified as a part of the NLM protocol, in order to allow implementation flexibility and to facilitate the development of new mechanisms without requiring the revision of related protocols.

The NLM uses the NSM protocol to enable it to recover from crashes of either the client or server host. To provide this functionality the NSM and NLM protocols on both the client and server hosts must cooperate.

The NSM is a service that provides applications with information on the status of network hosts. Each NSM keeps track of its own "state" and notifies any other NSM of a change in this state upon request. The state is merely a number which increases by one each time the state of the host changes.

Applications register the network hosts which they are interested in with the local NSM. If one of these hosts crashes, the NSM on the crashed host, after a reboot, will notify the NSM on the local host that the state changed. The local NSM can then re-send the request to recovery the lock which was lost when the host crashed. The detailed description for this will be discussed in the next chapter.

3.2 NSM RPC-based Procedures

The following specification summarizes the protocol used by the NSM using RPC language. The detailed data structures for NSM procedures are in appendix 1.

```

program SM_PROG {
  version SM_VERS {
    void SM_NULL(void) = 0;
    struct sm_stat_res SM_STAT(struct sm_name) = 1;
    struct sm_stat_res SM_MON(struct mon) = 2;
    struct sm_stat SM_UNMON(struct mon_id) = 3;
    struct sm_stat SM_UNMON_ALL(struct my_id) = 4;
    void SM_SIMU_CRASH(void) = 5;
    void SM_NOTIFY(struct stat_chg) = 6;
  } = 1;
} = 100024;

```

3.3 Semantics of NSM Operations

The following sections describe how each of the NSM remote procedures operates.

3.3.1 SM_NULL (Procedure 0)

By convention, procedure 0 in any RPC program is termed *null* because it does not perform any action. An application can call it to test whether a given server is responding.

3.3.2 SM_STAT (Procedure 1)

This procedure tests to see whether the NSM agrees to monitor the given host.

3.3.3 SM_MON (Procedure 2)

This procedure initiates the monitoring of the given host. This call enables the NSM to respond to notification of change of state calls (SM_NOTIFY) for the host specified in the arguments, and to notify that host, via the SM_NOTIFY call, when its state (that is, crash and reboot) changes.

When an NSM receives an SM_MON call it must save the name of the host in a notify list on the disk. If the host running the NSM crashes, on reboot it must send out an SM_NOTIFY call to each host in the notify list.

3.3.4 SM_UNMON (Procedure 3)

This procedure stops monitoring the host specified in the argument.

3.3.5 SM_UNMON_ALL (Procedure 4)

This procedure stops monitoring all hosts for which monitoring was requested.

3.3.6 SM_SIMU_CRASH (Procedure 5)

This procedure simulates a crash. The NSM releases all its current state information and reinitializes itself, incrementing its state variable. It reads through its notify list (see SM_MON) and informs the NSM on all hosts on the list that the state of this host has changed, via the SM_NOTIFY procedure.

3.3.7 SM_NOTIFY (Procedure 6)

When an NSM receives the SM_NOTIFY call it must search its notify list for the monitored host. The host will be found in the notify list if an SM_MON call was made to the NSM to register the host.

CHAPTER FOUR

System Analysis

4.1 A Concurrent Or An Iterative Design

The server designs can be divided into two categories: Those that handle requests iteratively and those that handle them concurrently.

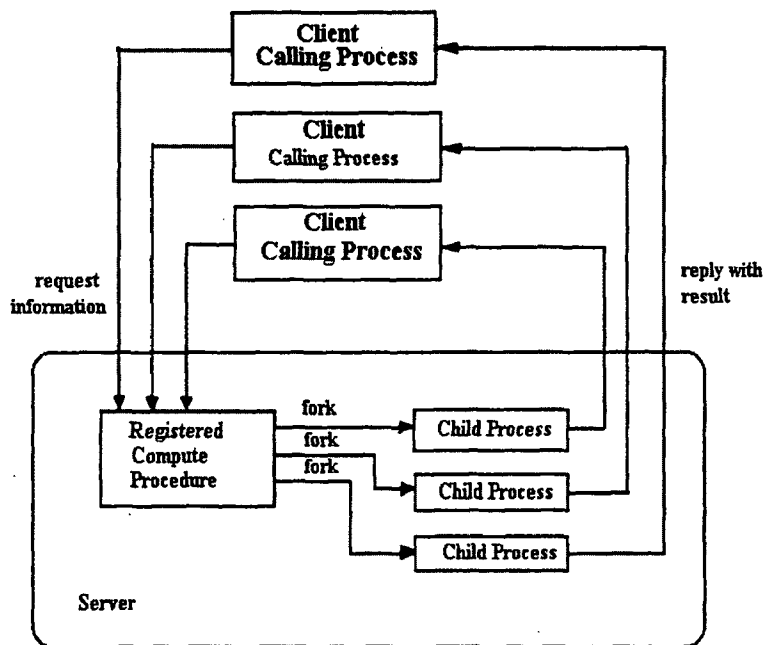


Figure 4.1. Multi-tasking via a concurrent server.

The concurrent server (Figure 4.1) is driven by the arrival of a request and not by the time slicing mechanism in the operating system. When a request arrives the server will

fork a child process to handle the request so that the server can continue to accept other requests without being blocked.

The iterative server (Figure 4.2) can only accept one request at a time. This means if there are two requests coming at the same time, one of them will be blocked until the server completely finishes the other one.

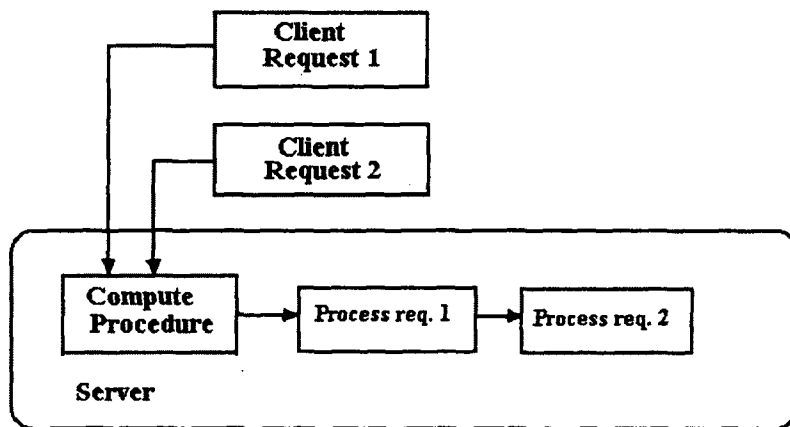


Figure 4.2. An iterative server handles two requests

In this project, the server of the NLM is designed as a concurrent server and the server of the NSM is designed as an Iterative server. The reasons for these two different designs will be discussed in more detail a little later.

4.2 Design A Reliable file Locking Service

A reliable file locking service has to provide at least two functions: *forever-lock* & *detecting and crash recovery*.

Forever-Lock Detecting

When a client program requests a file lock service from a server, the server will *fork* a child process to lock the file for the client and then this child process will go to sleep until an unlock request arrives and kills this sleeping process. But when the client program requests a lock and exits without releasing the lock first, the lock will be held by the server forever.

In order to solve this problem, the client program must have the ability to detect itself so that if the calling client exits without releasing the lock, this sleeping process can be notified and canceled. To do this, every time the client program request a remote lock service from the remote server, the client program has to register itself in the local server so that the local server can monitor the client program. If the client exits without releasing the lock, the local server will notify the remote server that the calling client no longer exists so the remote server can kill the sleeping process.

Another situation that will cause forever-lock is when the client host goes down before the client program releases a lock in remote host. In this case, the client host must have a way to notify the remote host to release all the locks requested by the client host.

To solve this problem, every time a client requests a lock from a remote server, the client must register the remote host name in its local server so that when the client host crashes and restarts, it will go through the remote host list (notify list) and notify each of the remote hosts in the list to remove all the locks requested by the client host.

Crash Recovery

When a server host crashes, all the locks that are requested by the client host will be lost. To recover these locks, the server program must save all the client requests on disk so that when the server host crashes and restarts, it can go through these requests and notify each individual client host to reclaim the locks again. Of course, the client host will reclaim the locks automatically without interrupting the application which is running on the client host.

4.3 A Better Design

SUN, the first developer of NLM and NSM, has long been known for its unreliable remote file locking service. One of the biggest problems is that when a process locks a remote file by using the UNIX system call *flock()*, its child process can not detect it. This means when a remote file is locked by a process and then if its child process issue a remote lock on the same file again, the system will reply a LOCK_GRANTED to the child process. In this case, a process can not guarantee to lock a file exclusively.

In this project, this problem has been solved by using *fcntl()*. Like *flock()*, *fcntl()* is a UNIX system call that can provide file locking but without the unreliable problem of *flock()*. When a client program requests a file locking using the system call *flock()* or *fcntl()*, the server always use *fcntl()* to lock the file for the client. Because of this small change, this program can provide more reliable network file locking service than SUN's program.

CHAPTER FIVE

System Design and Implementation

5.1 Implementation Approach

The target system contains two server programs, NLM and NSM, and one client program. The server programs work as *daemon* processes waiting for the request from the client program.

This system design is based on the SUN RPC so the RPCgen and some of the RPC library routines are used.

5.2 RPCgen

RPCgen developed by SUN is an RPC protocol compiler to help programmers to build a client/server communication model without worrying about the complicate networking programming. It reads the RPC protocol specification as an input and produces client and server program stubs that use lower-level RPC calls. RPCgen supports multiple transports and it generates a dispatch routine that is capable of handling multiple procedures and versions.

If you use RPCgen, you don't have to write the code for RPC communications in your client and server programs. Instead, these functions are performed by the client and server stubs that the RPCgen generates from your RPC protocol specification. These stubs include the code to marshal arguments, to send an RPC message, to dispatch an incoming call to the correct procedure, to send a reply, and to translate arguments and results

between the external representation and native data representation. However, the files that RPCgen produces do not form complete programs. They need to be customized and require some interface routines that the programmer must write. All these will be discussed in more detail in next section.

5.3 Designing the Server

5.3.1 The Six Steps To Build A Server

Figure 5.1 shows the input files required for RPCgen and the output files it generates. To create the required files and combine them into a NLM or NSM server, the following five steps have to be taken.

1. Write the RPCgen specification for the server program, including names and numbers for the remote procedures and the declarations of their arguments.
2. Run RPCgen to check the specification and, if valid, generate the four source code files that will be used in the server program.
3. Customize the server-side stub that is generated by RPCgen.
4. Develop each remote procedure in the RPC specification.
5. Develop the server interface routines.
6. Compile and link together the server programs.

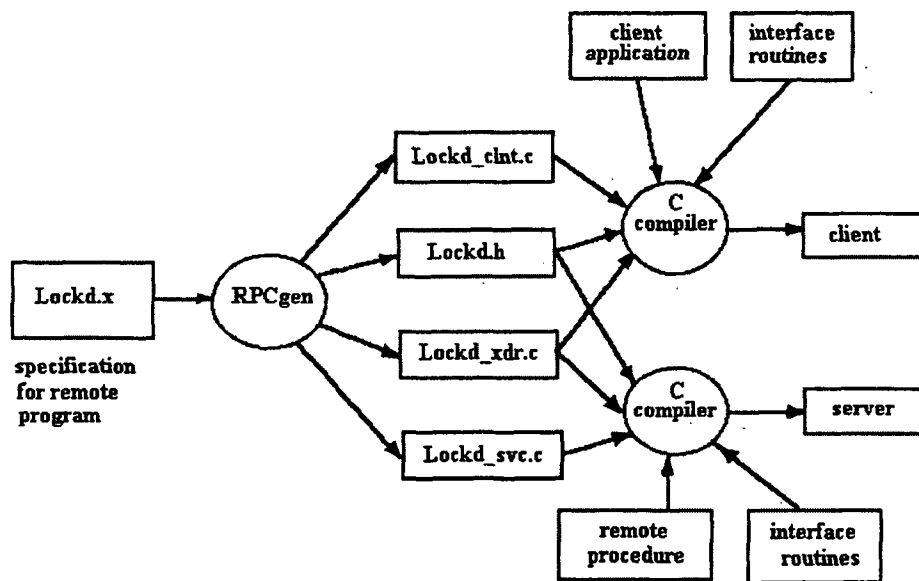


Figure 5-1. *The files required to build a client and server from the output of rpcgen, and the compilation steps required to process them.*

5.3.1.1 Step 1: Write An RPCgen Specification

In this project, the RPCgen specifications for NLM and NSM are based on SUN's NLM and NSM protocols and some private procedures are added for internal communication. The specifications are shown in chapter 2.2 and 3.2.

5.3.1.2 Step 2: Run RPCgen

After the specification has been completed, the next step is to run RPCgen to check for syntax errors and generate four files of code as Figure 5.1 shows.

RPCgen uses the name of the input file when generating the names of the four output files. For example, because the input file began with `lockd`, the output files will be named: `lockd.h`, `lockd_clnt.c`, `lockd_svc.c` and `lockd_xdr.c`.

- `lockd.h` is a head file containing all the user defined types and structures
- `lockd_clnt.c` is a client-side stub.
- `lockd_svc.c` is an server-side stub.
- `lockd_xdr.c` is an interface routine to communicate with XDR routines.

5.3.1.3 Step 3: Customize the Server Stub

In this project, the servers have to be *daemon* processes. So the server stub, `lockd_svc.c`, which is generated by RPCgen has to be customized to work as a *daemon* process. To make a *daemon* process, the following procedures have to be done:

- Close all open file descriptors
- Change the current working directory
- Reset the file access creation mask
- Disassociate from the process group
- Disable the control terminal

5.3.1.4 Step 4: Develop the Remote Procedures

Remote procedures are the kernel of a server. There are 19 remote procedures in the NLM server and 7 in the NSM server. All these have to be implemented to provide a complete lock service.

5.3.1.5 Step 5: Develop the Client Interface Routines

The files which are generated by RPCgen do not form complete programs. They require client-side and server-side interface routines to connect the stubs (generated by RPCgen) and the implementation programs.

On the client side, the interface routines accept calls from the server-side stub, and pass control to the procedure that implements the specified call. For each procedure in the server, we need an interface routine to translate from the argument types which are passed by server-side stub to the argument types that the called procedure use.

5.3.1.6 Step 6: Compile and Link Together the Server Program.

A server program consists of four main files: The remote procedures, the server-side stub (generated by RPCgen), the server-side interface routines, and the XDR procedures (generated by RPCgen). When all these files have been compiled and linked together, the resulting executable program becomes the server.

5.3.2 Implement NLM As A Concurrent Server

In a UNIX system, a process can hold a locked region only when the process is alive. This means if a process locked a region and terminated without releasing the locked region, the operating system will release the locked region automatically. In order to accept more requests from the clients without being blocked by the previous lock request, the NLM server has to be designed as a concurrent server.

The concurrent NLM server will *fork* a child process to handle the lock service when it receives a lock request from the client. The child process will lock the region for the client and then go to sleep forever. To do this makes the child process remain alive. (To remove the lock just simply kill the child process.)

5.3.3 Implement NSM As An Iterative Server

From the point of view of efficiency, an iterative server is more efficient than a concurrent server. For the NSM server, it is not necessary to work concurrently, so an iterative design is the best choose.

5.4 Designing The Client

5.4.1 Develop the Client Procedure

In this project, the client procedure consists of three functions: *ropen()*, *rflock()*, and *rfcntl()*. These three functions work as an interface between application programs and server programs.

ropen() - This function extends the UNIX system call *open()* to provide a table to indicate each file descriptor and its file path. It is necessary to do this because there is no UNIX system call that can retrieve the file path from a given file descriptor.

rflock() - This function works the same way as the UNIX system call *flock()* but works with *ropen()* instead of regular "open". When this function is called, it will check whether the given file descriptor is a remote file or a local file. If the

file is a local file, it will issue a system call *flock()*. Otherwise it will invoke the remote procedures in the target remote host.

rfcntl() - This function works the same way as the UNIX system call *fcntl()*, which provides a file region (record) lock services. Like function *rflock()*, it works with *ropen()* instead of regular "open" and also it will check whether the given file descriptor is a remote file or local file. If the file is a local file, it will issue a system call *fcntl()*. Otherwise it will invoke the remote procedure in the target remote host.

In addition to the three functions, the client program also contains three special macro definitions:

```
#define open(x,y)   ropen(x,y)
#define flock(x,y)  rflock(x,y)
#define fcntl(x,y,z) rfcntl(x,y,z)
```

With these three macro definitions, the programmer can use regular system calls *open()*, *flock()* and *fcntl()* to work with either a remote file or a local file, so that the whole system is transparent for a programmer. The preprocessor will translate these three system calls into the client functions *ropen()*, *rflock()* and *rfcntl()* before the program is compiled.

5.4.2 Develop the Client Interface Routines

On the client side, the original application program controls processing. It calls interface routines using the same procedure names and argument types as it originally used to call those procedures which have become remote in the distributed version. Each interface routine must convert its arguments to the form used by RPCgen, and must then call the corresponding client-side communication procedure.

5.4.3 Compile and Link Together the Client Program

A complete client program consists of four main files: The client procedure, the client-side interface routines, the client-side stub, and XDR procedures. When all these four files have been compiled and linked together, the resulting executable program becomes the client.

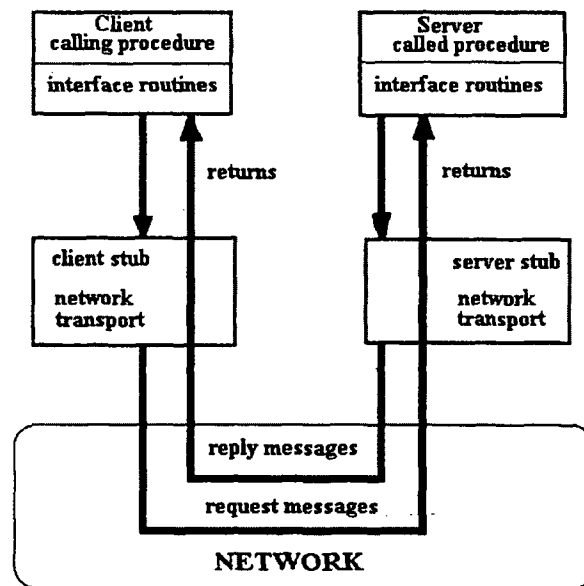


Figure 5-2. *Remote procedure call communication.*

5.5 The Interaction Between The NLM and The NSM

This section will describe the interaction between the NLM and the NSM for the synchronous procedures to show their interdependency.

Locking

The NLM_LOCK RPC requests may be blocking or non-blocking. When the server NLM receives the NLM_LOCK request, it must make a call to the SM_MON procedure on its local NSM to monitor the calling host. The SM_MON call includes the name of the host to be monitored and an RPC to be called if the NSM is notified of a state change (crash and restart) for the monitored host.

If the lock can be granted immediately, or the call was non-blocking, the RPC returns immediately with the appropriate status (granted or denied).

If the lock can not be granted immediately (it conflicts with an existing lock) and the call is a blocking call, the RPC will return with a blocked status, thus allowing the client NLM to continue processing. At this point the client NLM can choose to cancel the outstanding lock request by calling the NLM_CANCEL RPC. Upon reception of an NLM_CANCEL request, the server NLM should then delete the outstanding lock request, and may request its local NSM to stop monitoring the calling host by calling the SM_UNMON RPC.

When the blocked lock request can be processed, the server NLM makes an NLM_GRANTED call-back to the client NLM, indicating success or failure. Once the

lock has been granted, the client NLM instructs the local NSM to monitor the server via the SM_MON RPC.

Server Crash Recovery

When the server host crashes then restarts, its NSM will go through the notify list and will call the SM_NOTIFY procedure for each of these hosts to inform them of the state change. Each local NSM that receives this SM_NOTIFY call will search their notification list and make the corresponding RPC supplied in the previous SM_MON call, to the interested parties. One of the interested parties will be the client NLM protocol implementation that will have supplied an RPC that can go through the steps necessary to re-establish the lost locks during the NLM server's grace period. (the grace period in this project is 45 seconds)

Client Crash Recovery

If the client host crashes and then reboots the NSM will go through the same process notifying the NSMs on hosts in the notification list (via the SM_NOTIFY procedure call) that there was a change in state. The server NSM will receive this notification call and in turn notify the server NLM, via the provided RPC, that the client host had crashed. The server NLM can then dispose of all locks held by the crashed host.

Unlocking

A monitored lock is unlocked by making a call to the NLM_UNLOCK procedure. The server NLM will process the request and release the lock. The server NLM can then

ask its local NSM to stop monitoring the calling host via the SM_UNMON procedure call. At this point the server NLM will check existing blocked lock requests and service them if possible.

5.6 How RPC Systems Work

In general, services make themselves known to a network of clients through an independent naming service (**portmap daemon**). This service can give clients the address which is needed to open a communication channel with a server. The server can then accept (or deny) client requests and send replies as necessary. The connection is closed once the remote session is completed.

The following figure outlines the three steps necessary before an RPC client can call a server.

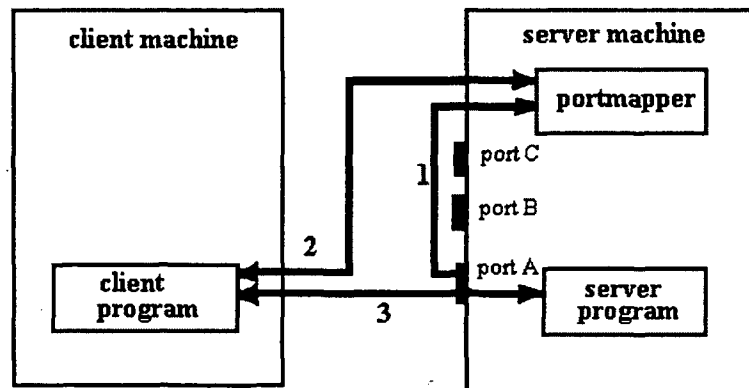


Figure 5-3. *Rpc client/server setup.*

1. When any RPC server (*daemon*) is started, it establishes an address where it listens for requests. It registers the port number (address) with the portmapper. It also registers the RPC program numbers and versions that the server is prepared to service. The client and server applications have arbitrary port numbers or addresses.
2. Before a client can make a remote procedure call (using a server program number), the portmapper of the server machine is consulted to identify the port number (address) that is to receive an RPC request message.
3. The client and server can now open a communication path to perform the remote procedure execution. The client makes its request; The server sends a reply.

5.6.1 RPC Library Calls

In this section, several RPC library routines which are used in this project will be discussed to show how to build a client and a server program by using the RPC library calls.

clnt_create()

clnt_create() create the client *handle* for the specified server host, program numbers, and version numbers (client-side).

clnt_call()

clnt_call() uses the client *handle* which is generated by *clnt_create()* to call the remote procedure (client-side).

svcadp_create()

svcadp_create() creates a UDP/IP-based RPC transport (server-side).

svctcp_create()

svctcp_create() creates a TCP/IP-based RPC transport (server-side).

svc_register()

svc_register() works directly with the **portmap** to register a dispatch routine of a specified protocol (server-side).

svc_run()

svc_run() is an indefinite loop waiting for the requests (server-side).

5.6.2 Using RPC Library Calls in a Program

The following figure illustrates a sequence of calls made by a client and a server using RPC library calls.

CLIENT SIDE

```
clnt_create()
  ↓
clnt_call()
```

SERVER SIDE

```
svctcp_create()
  ↓
svc_register()
  ↓
svculdp_create()
  ↓
svc_register()
  ↓
svc_run()
```

Figure 5.4. An example sequence of RPC calls made by a simple client and a server. The server runs in a loop (*svc_run()*), waiting for the request from the client.

The client uses *clnt_create()* to create the *handle* for the specified server host, program, and version number (transport protocol is selectable as TCP or UDP) then uses *clnt_call()* to connect with the server to request a remote procedure service by using the *handle*.

The server uses *svculdp_create()* and *svctcp_create()* to create a TCP and a UDP RPC service transports at a particular socket, and then uses *svc_register()* to register them with the portmapper, after that uses *svc_run()* to start the process of waiting on the end of a socket, listening for a valid connection. Once a request is accepted, the process branches to the named procedure (requested by the client) with the input (request) argument, and returns the result (reply).

CHAPTER SIX

Conclusion

This program is based upon a client/server network model using the RPC to provide a reliable file and record locking server over the NFS.

This program has been tested on SUN workstations. To port this program to other systems, minor changes in this program may be necessary. This is because some system calls used in this program for accessing a mounted file system are not universal standard. For example, *statfs()* that is a system call to returns information about a mounted file system is not identical between SUN and BSD environments. Nevertheless, my study showed that the RPC Library is not completely compatible between different system environments. In some situations, the communication will become unreliable between two different systems.

For further study of this project, one can investigate the RPC Library to find the problems which make the communication between different system unreliable.

BIBLIOGRAPHY

X/Open Company Limited. X/Open CAE Specification. Protocols for X/Open Internetworking: XNFS, Issue 4, 1992.

Douglas E. Comer. Internetworking with TCP/IP Vol. I. Prentice-Hall, 1991.

Douglas E. Comer, and David L. Stevens. Internetworking with TCP/IP Vol III. Prentice-Hall, 1993.

John Bloomer. Power Programming with RPC. O'Reilly & Associates, Inc., 1992.

W. Richard Stevens. UNIX Network Programming. Prentice-Hall, 1990. pp. 72-82.

ACKNOWLEDGMENTS

I would particularly like to acknowledge Dr. Youlu Zheng for his effort and support, and the initial idea for this project. His expertise is the technical backbone of this project.

I also would like to acknowledge Dr. James Ullrich and Dr. Steve Sheriff for their time and effort in providing me with invaluable help and suggestions.

APPENDIX 1

Data Structures For NLM

```
enum nlm_stats {
    LCK_GRANTED = 0,           /* call completed successfully */
    LCK_DENIED = 1,           /* request failed */
    LCK_DENIED_NOLOCKS = 2,   /* failed.(sever couldn't alloc. resources */
    LCK_BLOCKED = 3,          /* blocked. sever will make a call-back */
    LCK_DENIED_GRACE_PERIOD = 4 /* fail. sever has been rebooted */
};

struct nlm_stat {
    nlm_stats stat;
};

struct nlm_res {
    netobj cookie;
    nlm_stat stat;
};

struct nlm_holder {
    bool exclusive;
    int uppid;
    netobj oh;
    unsigned l_offset;
    unsigned l_len;
};

union nlm_testreply switch (nlm_stats stat) {
    case LCK_DENIED:
        struct nlm_holder holder; /* holder of the lock */
    default:
        void;
};

struct nlm_testres {
    netobj cookie;
    nlm_testreply test_stat;
};
```

```

struct nlm_lock {
    string caller_name<LM_MAXSTRLEN>;
    netobj fh;          /* identify a file */
    netobj oh;          /* identify owner of a lock */
    int uppid;          /* Unique process identifier */
    unsigned l_offset;  /* File offset ( for record locking) */
    unsigned l_len;     /* Length (size of record) */
};

struct nlm_lockargs {
    netobj cookie;
    bool block;          /* flag to indicate blocking behaviour. */
    bool exclusive;     /* If exclusive access is desired. */
    struct nlm_lock alock; /* the actual lock data (see above) */
    bool reclaim;       /* used for recovering locks */
    int state;           /* specify local NSM state */
};

struct nlm_cancargs {
    netobj cookie;
    bool block;
    bool exclusive;
    struct nlm_lock alock;
};

struct nlm_testargs {
    netobj cookie;
    bool exclusive;
    struct nlm_lock alock;
};

struct nlm_unlockargs {
    netobj cookie;
    struct nlm_lock alock;
};

/* the maximum length of the string */
const SM_MAXSTRLEN = 1024;

```

APPENDIX 2

Data Structures For NSM

```
/* sm_name is the name of the host to be monitored */
struct sm_name {
    string mon_name<SM_MAXSTRLEN>;
};

enum res {
    STAT_SUCC = 0, /* NSM agrees to monitor */
    STAT_FALL = 1 /* NSM cannot monitor */
};

struct sm_stat_res {
    res res_stat;
    int state;
};

struct sm_stat {
    int state; /* state number of NSM */
};

struct my_id {
    string my_name<SM_MAXSTRLEN>; /* hostname */
    int my_prog; /* RPC program number */
    int my_vers; /* program version number */
    int my_proc; /* procedure number */
};

struct mon_id {
    string mon_name<SM_MAXSTRLEN>;
    struct my_id my_id;
};
```

```
struct mon {  
    struct mon_id mon_id;  
    opaque priv[16];          /* private information */  
};
```

```
struct stat_chge {  
    string mon_name<SM_MAXSTRLEN>;  
    int    state;  
};
```

APPENDIX 3

Programs Installation

Server Programs Installation

To install the server programs which are daemon processes, you have to check your system menu to find out the proper directory for daemon programs then copy the server programs to that directory. After that, consider the following different ways a daemon process can be started.

1. Modify */etc/rc* file to start a daemon process. Most system daemons are started by the initialization script */etc/rc*, which is executed by */etc/init* when the system is brought up to multi-user mode.
2. Modify */usr/lib/crontab* file to start a daemon process. A standard UNIX process named *cron* performs periodic tasks at given times during the day, taking its instructions from the file */usr/lib/crontab*. Therefore by modifying this file, one can start a daemon process.
3. By executing the *at* command, which schedules a job for execution at some later time.
4. Start a daemon from a user terminal, as a foreground job or as a background job. This is usually done when testing a daemon.

After the server programs, *lockd* and *statd*, have been started, their "lifetime" is the entire time that the system is operating; usually they do not die and get restarted later.

They spend most of their time waiting for some event to occur at which time they perform their service.

Client Program Installation

The client program contains one header file (*lockd.h*) and one object file (*lockd.o*). To install the header file, you can simply copy it into */usr/include* directory. For the object file, you can use one of the following ways to install it.

1. Use the user's command *ar* to add the object file into the */usr/lib/libc.a* (library archive).
If you do this way, you don't have to explicitly link the object file (*lockd.o*) into your executable file.
2. Use the user's command *ar* to build a new library archive in */usr/lib* directory. If you do this way, you have to link this library archive explicitly when your application program is compiled. (Use option **-l**).
3. Copy the object file to */usr/lib* directory. If you do this way, you have to link this object file explicitly when your application program is compiled (Use option **-L**).

Write an Application Program

To use the lock functions provided by the client program, the header file *lockd.h* has to be included in the application program. With this header file, the programmer can use regular system calls *open()*, *flock()*, and *fcntl()* to work with either a remote file or a local file.

When the application program is compiled, a proper option may be needed (option **-I** or **-L**) depending on the way you install the object file (*lockd.o*).

Where are the programs ?

The source code and executable files are in **fjord.cs.umt.edu/~home/fjord/lin/project**.